Experimental Algorithms: Exercise sheet 2

Dr. Alexander van der Grinten

December 3, 2020

Notes

- Due by December 10, 2020, 9:00 AM.
- See the remarks on exercise sheet 1 (i.e., regarding the grading of exercises and on running your experiments).
- When adding new functionality to algorithms, add corresponding command line `--options`. In `experiments.yml` you can add more items to the `experiments` and/or `variants` sections to add new runs, see the documentation for more information.
- To generate plots, add a Jupyter notebook. You can copy the notebook from exercise sheet 1 and adapt it.

Exercise 4: Dynamic Hash Tables

In this exercise, we implement dynamically growing hash tables. Use the framework provided on Moodle as a baseline. Run the experiments on `gruenau3` or `gruenau4`.

(a) Implement a dynamically growing hash tables that uses linear probing (e.g., by adapting code from the static case). Start with 8 cells and double the number of cells whenever the fill factor is $> c$ for constant c (such that the fill factor after growing is $\geq \frac{c}{2}$).

Plot the performance of the dynamically growing table against the performance of the (already implemented) static table for 2^{26} insertions and $c \in \{0.5, 0.8, 0.9, 0.95, 0.99\}$. Use the same axis as on exercise sheet 1 (i.e., running time vs. $\frac{c}{2}$).

(b) Compare the (i) modulo and (ii) scaling strategies to select a cell of the hash table. As a reminder: so far, we have been using the modulo strategy that picks cell $(h \bmod m)$ (where h is the hash value and m is the size of the table). Scaling would pick $\lfloor \frac{h}{2^{31}} m \rfloor$ instead since we are using (signed) integers in the range $[0, 2^{31})$.

- What fraction of cache-misses can be avoided by using scaling?
- Add the scaling-based table to your plots from (a).

(c) Extend your hash table to use subtables. As in DySECT, use the highest bits of the hash function to select the subtable. Instead of doubling the size of the entire table, only double the size of subtables that fill beyond a fill factor of c.

Investigate how the number of subtables affects performance by plotting the running time of the algorithm vs. the number of subtables. Use $c = 0.9$ in this experiment. Perform experiments with $1, 2, 4, 8, 16$ and 32 subtables.

Exercise 5: Shared-Memory Parallelism

In this exercise, we consider the problem of counting all distinct length 4 substrings of a large binary string. For example, the string *aaaaabb* contains 3 distinct substrings of length 4, namely *aaaa* (which occurs twice), *aaab* and *aabb*.

Hint: substrings of length 4 can be uniquely mapped to 32-bit integers, using an expression such as:

```
unsigned int key =   (static_cast<unsigned int>(str[0]) << 24)
                   | (static_cast<unsigned int>(str[1]) << 16)
                   | (static_cast<unsigned int>(str[2]) << 8)
                   |  static_cast<unsigned int>(str[3]);
```

For this exercise, run the experiments on `gruenau1` or `gruenau2` (these machines have 36 CPUs and 72 hyperthreads).

(a) Implement a parallel algorithm to solve this problem using OpenMP. Use the framework provided in Moodle. You may use the concurrent hash table that is already implemented in `main.cpp`.[1]

Verify that your implementation returns correct results. For an input size of 2^{24} bytes (and the predefined random generator), the result is 16744462.

(b) Investigage the *scalability* (also called *strong* scalability) of your algorithm. To this end, plot the parallel speedup of the algorithm vs. the number of threads, for an input size of 2^{28} bytes. The speedup of the algorithm with t threads is defined as $\frac{time(1)}{time(t)}$ where $time(i)$ denotes the running time of the algorithm with i threads. Use $t \in \{1, 2, 4, 8, 16, 32\}$ threads (as in the supplied `experiments.yml`).

(c) Investige the *weak* scalability. In this case, the input size scales with the number of threads to account for the fact that adding more threads also adds more overhead. Use an input size of $2^{26} \cdot t$ bytes when you run the algorithm with t threads. As in (b), plot the parallel speedup.

Exercise 6: Concurrent Hash Tables

Download `ex2-hash-concurrent.zip` from Moodle. In the provided framework, implement the concurrent linear probing hash table from the lecture. Use 64-bit integer keys and values (i.e., `uint64_t`). You do not need to implement migration (a statically sized table is enough). Use the provided Simexpal configuration and Jupyter notebook to evaluate the performance of your implementation for $t \in \{1, 2, 4, 8, 16, 32\}$ threads on `gruenau1` or `gruenau2`. Remarks:

- To implement the algorithm, fill the empty methods in class `lockfree_linear_table`.
- Unfortunately, `std::atomic` does not support double-width CAS and we need to use "normal" integer variables together with OpenMP instead. Use the provided `dwcas()` function to perform double-width CAS. Use `#pragma omp atomic read` to perform atomic loads. Note that this pragma can only be used to read a single `uint64_t` atomically (but the algorithm does not rely on double-width atomic loads).
- You may assume that all keys are below $\leq 2^{63} - 1$. In particular, "negative" integers such as `uint64_t(-1)` do not occur as keys.
- Note: for testing purposes, it might make sense to lower `M` such that the random data is generated faster. However, make sure that you run the experiments with $M = 2^{27}$.

[1] However, this hash table does not (yet?) count the number of elements. If you want to add such functionality, make sure that your implementation is thread-safe.